GMXBUG User's Guide

Version 3.0

Copyright 1979 by
Gimix, Inc.
Chicago, Illinois

Table of Contents

GMXBUG monitor

The GMXBUG monitor is a 3K program which provides basic operating capabilities for a 6800-based microcomputer system. GMXBUG provides console I/O, tape cassette I/O, serial or parallel printer output, a powerful interactive debugger, and a package of utility routines.

GMXBUG differs from the other ROM monitors available for the 6800 in two major ways. First, no terminal is required. GMXBUG uses the Gimix video board for output, and connects directly to an ASCII keyboard for input. This arrangement is more flexible, less expensive, and many times faster than a terminal. Second, GMXBUG uses table-driven linkage for calls to the utility routines in the monitor. Table-driven linkage has many advantages over direct subroutine calls, including independence from absolute addresses and easier preservation of the machine registers.

The minimum system required by GMXBUG consists of the following:

Gimix 6800 CPU board
3k of PROM at $E000 containing GMXBUG
128 bytes of RAM at $A000
256 bytes of RAM anywhere between $0000 and $6FFF
Gimix 80 x 24 video board at $F000 - control registers at $F900
PIA at $8010 - port #4 - "A" side connected to ASCII keyboard

GMXBUG has facilities for saving and loading data on cassette tape. If the user wants to use these facilities, an ACIA at $8000 (port #0) connected to a cassette interface is required. GMXBUG also includes a driver for a hard-copy printer with either parallel or serial interface. The parallel driver will operate a printer connected to the "B" side of port #4; the serial driver will operate a printer connected to an ACIA at $800G (port #3).

MIKBUG compatibility

Nearly all of the 6800 software available in the hobbyist and personal computer area is written to run under the MIKBUG ROM monitor (trademark of MOTOROLA Inc.). In order to allow GMXBUG users to use this software, a 512-byte "cover area" at $E000 (the address of MIKBUG) has been included. In this cover area, all the important MIKBUG utility and I/O routines are implemented, either by included code, or by calls to the equivalent GMXBUG utilities. All of the following MIKBUG routines are supported by GMXBUG 3.0:

IO	E000	INHEX	E0AA
POWDWN	E005	IN1HG	E0BE
LOAD	E00A	OUT2H	E0BF
C1	E044	OUT2HA	E0C1
BADDR	E047**	OUT4HS	E0C8
BYTE	E055	OUT2HS	E0CA
OUTH1	E067	OUTS	E0CC

OUTHR	E06B	START	E0D0	
OUTCH	E075	SSBENT	E0D6	(Entry used by Smoke Signal DOS)
INCH	E078	CONTR1	E0E3	
PDATA2	E07B	INEEE	E1AC	
PDATA1	E07E	OUTEEE	E1D1	

Support of these calls makes GMXBUG compatible with nearly all the available 6800 software, with only minor modifications required.

** Under GMXBUG, the BADDR function is provided by a call to the GMXBUG HEXIN function. There are significant differences between the operation of HEXIN and the operation of BADDR-type routines in other monitors. Normally, BADDR accepts one key at a time, no backspacing is possible, if a non-hex character is entered the entry is aborted, and entry ends on the fourth digit automatically.

With HEXIN the GMXBUG line input function is used, then the entire line of input is converted to a 16-bit binary number which is returned in the index register. Only typing CR will end the entry. For further details see "Entering a tag" under "Debugger operation", and the description of the HEXIN utility.

GMXBUG reset routine

When the system is turned on, or the RESET button pushed, the system reset routine at $E397 is performed. This routine initializes the interfaces, pointers, and flags as given below:

Keyboard PIA: $2F is written to control register A

Cassette ACIA: reset & set to 8 data bits, 2 stop bits, no parity

Printer ACIA: reset & set to 8 data bits, 2 stop bits, no parity

Printer PIA: the data direction register is set to output, the CB2 output is strobed low, and $3E is left in control register B.

Stack pointer: set to point to the highest byte of memory not higher than $6FFF.

LINBUF: set to point to the start of the 256 byte page into which the stack pointer points.

Screen: cleared, and cursor goes to upper left corner

MMODE: set to 00 (off)

OUTPTR: set to 00 (screen only)

NULCNT: set to 6

DUPLEX: set to 00 (full duplex)

WAITON: set to 00 (off)

UPCASE: set to $FF (off)

PRFLAG: set to 00 (parallel type)

The reset routine is performed when the system is turned on, or the front panel RESET button is pushed. It can also be triggered by software: a Jump to the MIKBUG start ($E0D0), or to the GMXBUG reset address ($E38A), or system call 15 (RESET), or the "I" command in the debugger can also cause execution of the reset routine. However, the RESET line on the bus is not affected, and no hardware RESET is performed.

List of GMXBUG debugger commands

A	Hexadecimal arithmetic	P	print breakpoints
B	set breakpoint	R	register dump
C	checksum	S	save memory on tape
D	dump	T	test memory
E	erase	U	JSR to user PROM
F	formatted dump	X	block transfer
G	go to user program	Z	zap memory
H	hex locate		
I	initialize system values		
J	JSR to user program		
K	kill breakpoint		
L	load memory from tape		
M	examine and change memory		

+	increment memory pointer
-	decrement memory pointer
2	base 2 display
(CR)	leave memory mode
=	change byte
(sp)	change byte and increment pointer
"	enter ASCII characters

Function keys

ESC	abort entry	ctl-S	output stop/start
CR	complete entry		
ctl-H	backspace		

List of GMXBUG utility routines and codes

routine names and call codes in (hexadecimal) and decimal

XBA	(00)	00	INCHAR	(10)	16	LINEIN	(20)	32
XBAX	(01)	01	OUTCHR	(11)	17	MKWRIT	(21)	33
TXBA	(02)	02	TAPEIN	(12)	18	MKREAD	(22)	34
TBAX	(03)	03	TAPOUT	(13)	19	PCR	(23)	35
ADDAX	(04)	04	PSTRNG	(14)	20	PLF	(24)	36
ADDBAX	(05)	05	PSPACE	(15)	21	INECHO	(25)	37
SUBAX	(06)	06	PCRLF	(16)	22	INKEY	(26)	38
SUBBAX	(07)	07	PFFEED	(17)	23	PRINT	(27)	39
MULTAB	(08)	08	BREAK0	(18)	24	TBSRCH	(28)	40
PSHALL	(09)	09	BREAK1	(19)	25	ESCSND	(29)	41
PULALL	(0A)	10	BREAK2	(1A)	26	INCAPS	(2A)	42
STAALL	(0B)	11	BREAK3	(1B)	27			
HEXBIN	(0C)	12	PREGS	(1C)	28			
BINHEX	(0D)	13	PBYTE	(1D)	29			
MOVER	(0E)	14	HEXIN	(1E)	30			
RESET	(0F)	15	DEBUGR	(1F)	31			

GMXBUG video driver escape codes

POSX	01	(01)	SETM1	08	(08)	OUTSCR	15	(0F)
POSY	02	(02)	VDBLNK	09	(09)	OUTPRT	16	(10)
MEMOFF	03	(03)	VDNORM	10	(0A)	OUTBTH	17	(11)
MEMON	04	(04)	CBOFF	11	(0B)	SLOT0	18	(12)
SLCVID	05	(05)	CBON	12	(0C)	SLOT1	19	(13)
SLCFNT	06	(06)	CBSTDY	13	(0D)	PRGMOD	20	(14)
SETM0	07	(07)	CBFLSH	14	(0E)	TRMODE	21	(15)

GMXBUG scratchpad layout

	address	name	bytes	use
*	A000	IRQVEC	2	vector for IRQ interrupt
*	A002	BEGIN1	2	lower bound of a memory area
*	A004	END1	2	upper bound of a memory area
*	A006	NMIVEC	2	vector for NMI interrupt
*	A008	SP	2	
*	A00A	CKSM	1	
	A00B	SUBTOT	2	general purpose register
*	A00B	BYTECT	1	
*	A00C	XHI	1	
*	A00D	XLO	1	
*	A00C	MBTEMP	1	
	A00F	TEMP	2	general purpose register
*	A00F	TW	2	
*	A011	MCONT	1	
*	A012	XTEMP	2	
	A014	BEGIN2	2	lower bound of a second memory area
	A016	END2	2	upper bound of a second memory area
	A018	CURSOR	2	pointer to cursor position in video RAM
	A01A	WRAP	1	wraparound flag for video driver
	A01B	CHRSLT	1	mask for slot 0/slot 1 output
	A01C	ESCSTR	1	escape sequence counter
	A01D	ENDLIN	2	pointer to last byte of line in buffer
	A01F	LINBUF	2	pointer to start of the line buffer
	A021	MEMPTR	2	pointer to current byte in memory mode
	A023	MMODE	1	memory mode on/off flag
	A024	BRKTBL	16	table of data for 4 breakpoints
	A034	USRVEC	2	vector for user-defined SWI calls
	A036	OUTPTR	1	output device indicator flag
	A037	NULCNT	1	number of nulls inserted after CR by PRINT
	A038	DUPLEX	1	half/full duplex flag
	A039	WAITON	1	ctl-S on/off flag 00:OFF, FF:ON
	A03A	UPCASE	1	force upper case/enable lower case flag (0/$FF)
	A03B	ESCVEC	2	pointer to active escape routine
	A03D	SWIFLG	1	SWI control flag
	A03E	PRFLAG	1	parallel/serial printer flag → 0:PAR FF:SER
	A03F	DURATN	2	duration of BELL character tone
	A041	PERIOD	1	period of BELL character tone (4025)
*	A048	PCLOCN	2	jump address for user programs

* These locations are used the same in GMXBUG and MIKBUG.

GMXBUG debugger operation

Upon entry to the debugger, GMXBUG displays ">". This is the GMXBUG prompt, indicating that the user may enter a command.

To enter a command, type the letter for that command. For example, to perform the R command, type "R" on the keyboard.

Some commands require information such as addresses or byte values; these numbers are entered in hexadecimal and are referred to as tags.

Entering a tag

Tag values are entered through the GMXBUG line input utility, and then converted to binary with the GMXBUG hex-to-binary conversion utility. To enter a tag value, type the number, then type carriage return (CR) to terminate entry. This CR will not be performed. The value received by GMXBUG will be the value of the four rightmost digits to the left of the first non-hexadecimal character in the entry. If there are less than four digits, the entry is zero-filled from the left. If no digits are typed, the entry is 0000. If more than four digits are typed, all but the rightmost four are ignored. During entry, the user may correct a typing mistake by hitting backspace. See the description of the line input utility for the use of this key. Also available during entry is ESC; typing ESC aborts the command.

```
examples:  1234(CR)      enters 1234
           11111234(CR)  enters 1234
           +1234(CR)     enters 0000  (+ stops conversion)
           22 55         enters 22    (" " stops conversion)
           QRWKL(CR)     enters 0000  (Q stops conversion)
           1E(CR)        enters 001E  (zero filled)
           (CR)          enters 0000  (zero filled)
           123(ESC)      aborts command
```

Certain commands (C, S, T) require two tags. Both tags are entered on the same line. When the user types CR to complete entry of the first tag, GMXBUG does not echo the CR. Instead, a space is displayed, to prompt entry of the second tag. No user-typed spaces are necessary. When the CR to complete the second tag is typed, then the command is performed. At any time during the typing of either tag, ESC may be typed to abort the command. Note: once the CR to complete tag1 is typed and entry of tag2 is begun, backspace will not move the cursor past the beginning of tag2. Tag1 can not be corrected once the CR is typed. If tag1 is in error, the command must be aborted.

```
examples: 1234(CR) 4567(CR)  (enters 1234 and 4567)
          1(CR) 23(CR)       (enters 0001 and 0023)
          1A2B(CR) 9E3(ESC)  (aborts command)
```

The X and Z commands require three tags; these commands work the

same as C, S, and T, except that after the second tag is completed, a third tag must be entered and the second tag is fixed.

The H command requires four or five tags; if tag4 is a CR with no digits, then tag5 will be skipped. Otherwise it is the same as X, Z, C, S, and T.

I command entry format

The I command requires a special type of input. To select the system value to initialize, the user enters a single letter without CR. Then, depending on the type of value, the user must enter either a tag or another letter.

Aborting a command

When ESC is typed during the entry of a tag or other data, or if an illegal tag value is entered, the command is aborted. All data entered is ignored, and GMXBUG starts a new line and prints a ">". ESC does not cause GMXBUG to leave memory mode.

GMXBUG command descriptions

A hexadecimal arithmetic

Format: A tag1 tag2

The values of (tag1+tag2) and (tag1-tag2) are calculated and displayed on the same line.

```
example: >A 1123 203E 3161 F0E5
example: >A 1234 E 1242 1226
example: >A 8089 80B2 013B FFD7
```

B set breakpoint

Format: B tag1 tag2

Sets a breakpoint at tag1. Tag2 is the number of the breakpoint set, and must be less than 4. For further details, see "GMXBUG breakpoint system". Note: address of 0000 in the breakpoint table indicates a breakpoint not set. Therefore setting a breakpoint at 0000 is not allowed. If the value of tag1 is 0000, the command is automatically aborted.

```
example: >B 1234 0    (sets breakpoint 0 at $1234)
example: >B 2340 3    (sets breakpoint 3 at $2340)
example: >B A5 2      (sets breakpoint 2 at $00A5)
example: >B 00        (00 entry aborts command; no effect)
```

C checksum

Format: C tag1 tag2

The 24-bit sum of the 8-bit values contained in the bytes from tag1 through tag2 inclusive is calculated, and the result is displayed on the same line. This is useful for checking large blocks of code or data for accidental changes. Note: this command causes a read of every address in the specified range. Therefore, the range should not include I/O interfaces or the Gimix CPU board timer.

example: >C 1234 1FFF 02EF52
example: >C 0 0FFF 031DE1

D dump

Format: D tag1

Causes a combined hex and ASCII dump of memory starting at tag1. The dump is displayed one line at a time, 8 bytes per line. An additional line is displayed when any key except ESC is pressed. The dump ends on ESC. The bytes dumped are displayed as 2-digit hex numbers, and as their ASCII equivalents. The ASCII equivalents of control characters are displayed as graphics elements; values greater than $7F are displayed as specified by the programming of the video board. If console output is appearing on the printer, all control or other non-printable characters are printed as ".".

Note: this command uses BEGIN1, so if BEGIN1 is dumped the result is peculiar.

example: >D 1234
1234 01 12 23 34 45 56 67 78 /..#4EVGX/
123c 11 22 11 33 11 64 11 75 /.".3.d.u/

E erase

Format: E

Performs a form feed, moving cursor to upper left corner and clearing screen.

example: >E

F formatted dump

Format: F tag1

Causes a "disassembly" dump of memory starting at tag1. The dump is displayed one line at a time, with 1, 2 or 3 bytes per line. $3F (SWI) is handled like a 2-byte instruction, because of the special use of SWI in GMXBUG. As with the D command, additional lines are displayed on any key except ESC, which terminates the dump.

```
example: >F 1234
         1234 BD E1 AC
         1237 3F 10
         1239 43
         123A 7E E0 E3
```

G go to user program

Format: G tag1

Causes execution to begin at tag1. A special case exists for resuming execution after a breakpoint: for this, type "X(CR)" as tag1. This causes GMXBUG to execute an RTI. See "GMXBUG breakpoint system" for exact details.

```
example: >G 1234    Jump to 1234
example: >G X       exit from breakpoint
```

Another special case exists for programs which run under MIKBUG: such programs often store a start or restart address in $A048-$A049 for use by the MIKBUG G command. The GMXBUG G command can also use these addresses; this is done by typing CR instead of a tag. this will cause GMXBUG to jump to the address stored in $A048-A049. This can often save typing, but be sure the program does set up such an address before using this feature.

```
example: >G (CR)  go to address stored in $A048-A049
```

H hex locate

Format: H tag1 tag2 tag3 tag4 (tag5)

Searches through memory from tag1 thru tag2 for 1, 2, or 3 bytes equal to the tag3, tag4, and tag5 values. If a match is found, the address of the matching bytes is displayd on the same line. The match address is also stored in MEMPTR, so if memory mode is on, the first match byte becomes the new current byte immediately.

The user may choose to search for 1, 2, or 3 bytes. A null string (CR with no digits) means that byte and the following bytes (if any) are not to be matched for. Therefore, a null string may be entered to set a starting or ending address of 0000, but not a match byte of 0. If a null string is entered for tag3, then no bytes are to be matched, so the command aborts. If a null string is entered for tag4, then tag5 is also not to be matched, so tag5 entry is skipped.

```
example: >H 2000 2100 7E E1 AC 223D          (3-byte search)
example: >H 0 FFF 86 0D (CR)                 (2-byte search,
                                              no match)
example: >1000 34 H 100 23FF 39 (CR) 0447    (1-byte search)
         >0447 39
example: >237E 45 H 3000 3800 47 4F (CR)     (2-byte search,
         >237E 45                             no match)
```

I initialize system values

Format: I char1 (char2 or tag2)

Initializes a selected system value. Char1 indicates the value to be initialized; (char2 or tag2) is the value to initialize with. The values initialized by various letters, and the possible entries for each are given in the table below.

char1	2nd item	function
C	(Y N)	enable (Y) or disable (N) ctl-S function
D	(F H)	set duplex to full (F) or half (H)
N	tag2	set printer driver null count to tag2
O	(S P B)	set output device flag to screen (S), printer (P), or both (B)
P	(P S)	set printer type flag to parallel (P) or serial (S)
R	none	display current value of various flags
S	tag2	set stack pointer to tag2
U	none	initialize everything (Jump to reset address)

```
example: >I C Y      (turn on ctl-S)
example: >I D F      (set duplex to full)
example: >I N 11     (set null count to hex 11)
example: >I O P      (route console output to printer)
example: >I P P      (set printer type flag to parallel)
example: >I S 3FFF   (set stack pointer to $3FFF)
example: >I U        (jump to reset address)

example: >I R D=F O=S S=N N=06 P=S
         C0 53 53 E552 E59C 6FEF

    (Report status:  duplex=full,  output=screen,  ctl-S=off,
     null count=6, printer=serial, second line=register dump)
```

J JSR to user program

Format: J tag1

Stores a return address on the stack and starts execution at tag1. The return address is the address of a branch to the beginning of the debugger. If the routine at tag1 ends with an RTS, then control will automatically return to the debugger at the end of the routine.

example: >J 1234

K kill breakpoint

Format: K tag1

Removes breakpoint tag1. As in the B command, tag1 must be less than 4. For further explanation, see "GMXBUG breakpoint system".

example: >K 1 (kill breakpoint 1)

L load from tape

Format: L tas1

Causes a MIKBUG-tm format tape file to be read into memory. For details of the MIKBUG format, see "MIKBUG file format". If a zero value is entered, the file will be read normally. If a non-zero value is entered, the data will be stored in a continuous block starting at tas1, regardless of the addresses stored on the tape. For each record read in, a "#" is displayed. If a checksum error occurs, the address contained in the offending record is displayed with a "?", and the load continues.

example: >L (CR)#########
(normal load, no errors found)
example: >L 100###################
(data stored starting at $100, no errors)
example: >L (CR)#####0640?#####
(normal load, error in record with address $640)

M examine and change memory

Format: M tas1

Sets the memory mode flag (MMODE - $A023) to $FF, turning on memory mode, and stores the tas1 value in the memory pointer (MEMPTR - $A021-$A022).

Memory mode

When memory mode is on, the debugger begins each line with ">", then prints the value of the memory pointer and the contents of the byte it points to (the current byte).

example: >1234 56 where the memory pointer value is $1234
and the contents of byte $1234 are $56.

Memory mode enables six extra commands.

Note: unlike some other monitors, all GMXBUG commands are available while the user is examining and changing memory.

example: >1234 56 C 11DB 1235 000D32
>1234 56

The six additional commands available in memory mode are:

+ increment memory pointer

Format: >1248 EC +

Adds one to the address in MEMPTR.

example: >1234 56 +
         >1235 67

\- decrement memory pointer

Format: >1248 45 -

Subtracts one from the address in MEMPTR.

example: >1234 56 -
         >1233 78

2 base 2 display

Format: >1248 CE 2

Displays the contents of the current byte in base 2 (binary).

example: >1234 56 2 01010110
         >1234 56

(CR) leave memory mode

Format: >1248 CE(CR)

Stores 0 in MMODE, disabling memory mode functions.

example: >1234 56(CR)
         >

= change byte

Format: >1248 CE = tas1

Stores the less significant byte of tas1 in the current byte.
Note: this is useful for manipulating ACIAs or PIAs directly.

example: >1234 56 = 78
         >1234 78

(sp) change byte and increment pointer

Stores the less significant byte of tas1 in the current byte and increments the memory pointer.

example: >1234 56 78
         >1235 67

" enter ASCII characters

Format: >1248 CE " (characters)

Stores the ASCII characters following the " in successive bytes starting at the current byte, and increments MEMPTR for each character stored. Terminates on ESC or CR; if the first character is ESC or CR, no characters are stored and MEMPTR

is unchanged.

example: >1234 56 " ABCD
         >1238 9A

P print breakpoints

Format: P

Displays the addresses at which the 4 breakpoints are set. The number on the left is the number of the breakpoint; the number on the right is the address at which it is set. An address of 0000 indicates a breakpoint not set.

example: >P
     00 1234
     01 4EA2
     02 0678
     03 0000

where breakpoint #0 is set at $1234, #1 is set at $4EA2, #2 is set at $0678, and #3 is not set.

R register dump

Format: R

Displays the top 7 bytes on the stack and the stack pointer. If the debugger was entered through a breakpoint, the 7 bytes will be the machine register values saved by the SWI. The values are printed in the following format:

CC ACCB ACCA IX PC SP

example: >R
         D0 12 34 5678 9ABC 3FF0

S save memory on tape

Format: S tag1 tag2

Stores the contents of the bytes from tag1 through tag2 inclusive on tape in MIKBUG-tm format. 60 null characters are written to the tape at the start for leader. For details of the MIKBUG format, see "MIKBUG file format". Note: no device control characters are supplied; the user must start and stop the recorder by hand. The recorder must be going when the CR for tag2 is typed.

example: >S 1200 1AFF##################################

T test memory

Format: T tas1 tas2

Performs cyclic check on memory from tas1 through tas2 inclusive. This test is repeated through 256 passes to provide complete conversence testing. After each pass, a "#" is displayed. If an error is detected, a register dump will be displayed; for example,

D1 08 04 1234 E270 3FF7

The second number in the register dump (08) is the number of passes completed in hexadecimal, the fourth number (1234) is the address in hexadecimal where the defect was found, and the third number (04) is the error pattern. This pattern, when converted to binary, indicates which individual bits were in error: the binary value of 04 is 00000100, indicating that the 3rd bit was in error. Then the defective memory IC can be identified.

Note: this test may be performed on any size memory area, from 2 to 65536 bytes, starting at any address, not just 1k or 4k segments. However, it destroys the contents of the tested area. Therefore, don't test $A000 through $A00E or you will destroy the pointers for the routine, causing it to bomb. Also, do not test the area where the stack is stored; this will destroy the stack and bomb the routine.

example: >T 1234 5678
```
########################################################
########################################################
########################################################
########################################################
```
(no errors found)

example: >T 1000 16FF
```
###############
D1 0F 04 13F5 E270 3FF5
```
(3rd bit in byte 13F5 is bad; found after 15 passes)

example: >T 2000 2880
```
###################
D1 13 80 2800 E270 3FF3
```
(8th bit in byte 2800 is bad; found after 19 passes)

U JSR to user PROM

Format: U (CR)

Same as J EC00. Starts execution at $EC00, and leaves a return address on the stack. The return address is the address of a branch to the beginning of the debugger. If the

user routine at $EC00 ends with an RTS, control will automatically return to the debugger at the end of the routine. The JSR will not be executed until the user types CR.

X block transfer

Format: X tas1 tas2 tas3

Transfers the contents of memory from tas1 through tas2 inclusive to memory starting at tas3. If the source area and the destination area overlap, the move will be done in reverse order if needed.

example: X 1000 14FF 2000
example: X 3400 36FF 3600 (reverse move)

Z zap memory

Format: Z tas1 tas2 tas3

Fills memory from tas1 through tas2 with the value of the less significant byte of tas3.

example: Z 100 1FF 41
example: Z 380 3FF 0

GMXBUG breakpoint system

One of GMXBUG's most powerful facilities is the GMXBUG breakpoint system. This system provides the user with four independent breakpoints.

What is a breakpoint?

A breakpoint is a special instruction, available on most computers, which is provided as a debugging aid. The effect of a breakpoint is to halt execution of the current program, save the machine state (register contents, etc.), and transfer control to a debugger program. The programmer can then examine and modify the saved machine state, and resume execution if desired, all transparent to the current program.

The Motorola 6800 instruction set has the software inter- rupt instruction (SWI) as its breakpoint instruction. SWI saves the registers and jumps to the address in $FFFA.

GMXBUG extended breakpoints

GMXBUG uses the SWI instruction as a general-purpose monitor call; the byte after each SWI is interpreted as a modifying code, thus creating a set of 256 pseudo-instructions. These are the GMXBUG monitor calls. Four of them have been defined as breakpoints: 24 ($18), 25 ($19), 26 ($1A), and 27 ($1B).

To set a breakpoint requires four bytes of storage, two for the address at which it is set, and two for the code which is taken out when the breakpoint is inserted. The 16 bytes required by four breakpoints is allocated at $A024 as a table with four entries, called BRKTBL. The first two bytes of each entry store the address, and the other two store the saved code.

If a breakpoint is killed, then the table address for that breakpoint will be set to 0000 to indicate this. For this reason setting a breakpoint at 0000 is not permitted by GMXBUG.

Note: the reset routine has no effect on BRKTBL or the status of any breakpoint. BRKTBL will contain power-up garbage until the breakpoints are set or killed.

Setting a breakpoint

To set a breakpoint, use the "B" command in the

debugger. type "B", then enter the address for the breakpoint (see "GMXBUG debugger operation - tag entry"), then enter the number of the breakpoint. GMXBUG then puts the address and the two bytes of code from that address and the next byte in the BRKTBL entry for that breakpoint, and puts the system call ($3F) and the code for that breakpoint ($18, $19, $1A, or $1B) at that address and the following byte.

It is possible to set the same breakpoint a second time without killing the first setting. If breakpoint 2, for example, were set a second time, then the address and saved code in entry 2 of the breakpoint table would be lost, and the $3F-$1A already in memory would remain. Thus, it would no longer be possible to restore the code and delete the $3F-$1A except by hand. Worse, if the old breakpoint should happen to be executed, any attempt to kill it and resume execution would have unpredictable and possibly disastrous results. ALWAYS! kill a breakpoint before re-using it.

Listing the breakpoints

The "P" command may be used to display the status of the four system breakpoints. When "P" is typed, GMXBUG prints the numbers of the breakpoints and the addresses at which they are set.

```
example:  >P
          00 1234
          01 0246
          02 A9B8
          03 0000
```

Execution of a breakpoint

When a breakpoint is encountered during execution of a program it has the following effect:

As with other system calls, the registers are saved in the stack, and the call processing routine is performed. The call processing routine vectors all four breakpoints to BRKPRT ($E77C). BRKPRT prints "$$" and the breakpoint number, and a register dump, then jumps to the debugger.

```
example:  $$01
          D1 01 23 4567 89AB 3FF5
          >
```

Removing a breakpoint

To remove a breakpoint, use the "K" command in the debugger. Type "K", then type the number of the breakpoint to

be removed. The K command has varying effects, depending on whether the breakpoint is actually set, and whether it was executed. First, the debugger checks at the address in BRKTBL for the selected breakpoint to see if it is actually there; if so, the two bytes of saved code are restored. Next, it checks the 6th and 7th bytes in the stack to see if they match the address; if they match then the breakpoint was executed, and the 6th and 7th bytes are the saved PC value, so GMXBUG subtracts two from that value so that execution will resume with the restored code and not two bytes down. Finally, the BRKTBL entry for that breakpoint is set to all 00s, to indicate that it is not set.

Resuming execution after a breakpoint

To resume execution after a breakpoint, use the "G" command mand in the debugger. Type "G", then instead of an address, enter "X" for the tag. GMXBUG will perform an RTI, resuming execution of the program at the address at which the breakpoint was set. The executed breakpoint must be removed before execution can resume, but other breakpoints at other addresses may be left in place.

Use of system call 31 (DEBUGR) as a breakpoint

The system call to the debugger can be used similarly to the four breakpoint system calls. If this call is executed in a user program, the user may examine and change memory, including the saved resisters, use any other debugger facility, and resume execution with "G X", provided the stack pointer and the saved PC have not been modified. Since DEBUGR is a regular system call execution resumes with the byte after the call code; it is not set up or removed by the "B" and "K" commands. It may be left in place when execution is resumed.

GMXBUG monitor call system

GMXBUG uses the SWI (software interrupt) to call a GMXBUG routine from a user program or from inside GMXBUG. The SWI instruction saves the machine registers on the stack and jumps to the address at $FFFA, which is $E185 (SWIENT). This is the monitor call processing routine.

The SWIENT routine first tests SWIFLG ($A03D). If bit 8 of SWIFLG is set, then user SWI calls are in control, so SWIENT jumps to the address in USRVEC ($A034). If bit 8 is clear, then SWIENT gets the PC value from the stack. This value is the address of the code byte following the SWI.

Then the code byte is checked: if the code is 00 through 45, SWIENT uses the code as an index to get the address of the called routine from the table of addresses at $E1EA (SWITBL). If the code is 46 through 255, SWIENT jumps to the address in USRVEC. This allows the user to define codes 46 through 255.

Then SWIENT loads ACCA, ACCB, and IX with the values saved by the SWI, and jumps to the address. Thus, ACCA, ACCB, and IX have the same values when the SWI is performed and when execution of the called routine begins, making the call processing transparent. Also, the saved PC value is incremented, so that execution will resume with the instruction following the code byte upon return from the called routine.

The return from a GMXBUG routine to the calling program is done by means of the RTI (return from interrupt). This instruction loads the registers from the stack. All the GMXBUG routines end with RTI. Any user routine called by means of the SWI monitor call must also end with RTI.

Utility package

The utility package is a group of routines invoked by an SWI instruction followed by a one-byte code which determines the routine to be used (see "GMXBUG monitor call system" for details). The package is used extensively by the GMXBUG monitor and is also available to user programs.

Except where specifically stated, the contents of the machine registers are not altered by any utility routine. This means that the invoking program does not have to save and reload registers that it wants preserved, allowing compacter and more efficient programs.

The routines in the utility package fall into four categories:

(1) register and arithmetic functions
(2) I/O functions
(3) debugger functions
(4) miscellaneous routines

The register and arithmetic functions provide the user with a useful set of general-purpose register operations that can not be conveniently performed directly.

The I/O functions provide the user with console input and output, cassette input and output, and instant output of a space, a carriage return-line feed, a form feed, or a character string.

The debugger functions provide the user with a set of powerful tools for developing and debugging programs, including four independent breakpoints, register dump, and hexadecimal input and output. The debugger itself may also be invoked as a function.

Miscellaneous functions include saving and loading MIKBUG format files on cassette tape, a general-purpose line input routine, extra "carriage control" functions, an auxiliary printer driver, a keyboard scan function, and a table search function.

Calling a utility

The following is an example of how to set up a call to a utility routine in an assembler program.

```
* get item from table - using ADDAX
*
GETITM   LDX    #TABLE     load IX with address of start of table
         LDA A  INDEX      load ACCA with pointer to item
         SWI               jump to monitor
         FCB    8          8 is code for ADDAX
*                          ADDAX adds the pointer to the start of
*                          the table to get the item's address
         LDA A  0,X        get item from table
         RTS               end of subroutine
```

Register functions

name of routine	code	description
XBA	00	Exchange ACCA with ACCB Transfers the contents of ACCA to ACCB and the contents of ACCB to ACCA.
XBAX	01	Exchange IX with ACCA and ACCB Transfers the high byte of IX to ACCB, the low byte of IX to ACCA the contents of ACCB to the high byte of IX, and the contents of ACCA to the low byte of IX.
TXBA	02	Transfer IX to ACCB and ACCA Loads ACCB with the high byte of IX, and loads ACCA with the low byte of IX. The former contents of ACCB and ACCA are lost.
TBAX	03	Transfer ACCB and ACCA to IX Loads the high byte of IX with the contents of ACCB, and loads the low byte of IX with the contents of ACCA. The former contents of IX are lost.
ADDAX	04	Add ACCA to IX Adds the contents of ACCA to the contents of IX, and places the result in IX. The addition is unsigned. The former contents of IX are lost.
ADDBAX	05	Add ACCB and ACCA to IX Adds the contents of ACCB and ACCA to the contents of IX, and places the result in IX. ACCB and ACCA are used as the high and low bytes of a 16-bit operand. The addition is unsigned. The former contents of IX are lost.
SUBAX	06	Subtract ACCA from IX Subtracts the contents of ACCA from the contents of IX, and places the result in IX. The subtraction is unsigned. The former contents of IX are lost.
SUBBAX	07	Subtract ACCB and ACCA from IX. Subtracts the contents of ACCB and ACCA

from the contents of IX, and places the result in IX. ACCB and ACCA are used as the high and low bytes of a 16-bit operand. The subtraction is unsigned. The former contents of IX are lost.

MULTAB 08 Multiply ACCA and ACCB

Multiplies the contents of ACCA by the contents of ACCB, and places the high byte of the result in ACCB, and the low byte in ACCA. The multiplication is unsigned. The former contents of ACCA and ACCB are lost.

PSHALL 09 Push all registers on stack

Stores IX, ACCA, ACCB, and CC in the next five bytes below the top of the stack, and moves the stack pointer down five bytes. The order in the stack is, from the top, CC ACCB ACCA IXH IXL.

PULALL
10 $0A Pull all registers from stack

Loads CC, ACCB, ACCA, and IX from the top five bytes on the stack, and moves the stack pointer up five bytes. The bytes are taken from the stack and loaded into the registers in this order: CC, ACCB, ACCA, IXH, IXL. This function is the reverse of PSHALL.

STAALL 11 $0B Store all registers

Stores the contents of ACCB, ACCA, the high byte of IX, and the low byte of IX in the second, third, fourth, and fifth bytes in the stack. This routine is used by other utility routines to return the current contents of ACCB, ACCA, and IX to the calling routine. It should only be used in a routine invoked through the GMXBUG monitor call system.

Arithmetic functions

HEXBIN 12 $0C Hexadecimal-to-binary conversion

Converts a string of ASCII hexadecimal characters starting at the address in IX to a 16-bit binary number with the high byte in ACCB and the low byte in ACCA. Conversion is from left to right, and stops on the first non-hexadecimal character. The result is zero-filled from the left; see "Entering a tag" for examples.

BINHEX 13 $0D Binary-to-hexadecimal conversion

Converts the binary contents of ACCA into two hexadecimal ASCII digits, with the more significant digit in ACCA, and the less significant digit in ACCB.

Miscellaneous functions

MOVER 14 $0E General purpose block mover routine

Moves the contents of memory starting at the address in BEGIN1 through the address in END1 inclusive to memory starting at the address in BEGIN2. Checks for overlap of source and destination areas; if necessary the move is done back-to-front.

RESET 15 $0F System reset

Jumps to system reset address through the monitor linkage system. Note: this call does not return to the calling routine.

I/O functions

INCHAR 16 $10 Input character

Turns on the video board cursor and waits for keyboard input. When a key is entered, the cursor is turned off, and the input byte is returned in ACCA.

OUTCHR 17 $11 Output character

Outputs a character to the video board. For complete description, see "GMXBUG video driver".

TAPEIN 18 $12 Tape input

Inputs a byte from the ACIA at $8000. The data byte is ANDed with $7F and returned in ACCA.

TAPOUT 19 $13 Tape output

Outputs a byte through the ACIA at $8000.

PSTRNG 20 $14 Print string

Sends a string of bytes pointed to by IX to OUTCHR. The string must be terminated by a byte containing 04.

PSPACE 21 $15 Print space

Sends a space to OUTCHR.

PCRLF 22 $16 Print CR-LF

Sends a carriage return and line feed to OUTCHR.

PFFEED 23 $17 Print form feed

Sends a form feed to OUTCHR.

Debugger functions

BREAK0 24 $18
BREAK1 25 $19
BREAK2 26 $1A
BREAK3 27 $1B Breakpoints

See "GMXBUG breakpoint system".

PREGS 28 $1C Print registers

Displays the contents of CC, ACCA, ACCB, IX, PC and SP in hexadecimal. The format of the registers is

CC ACCB ACCA IX PC SP.

PBYTE 29 $1D Print byte

Displays the contents of ACCA as a two-digit hexadecimal number.

HEXIN 30 $1E Hexadecimal input

Inputs a string of hexadecimal characters from the keyboard, converts the string to a sixteen-bit binary number, and returns this number in IX and in ACCB and ACCA. The high byte is returned in ACCB, and the low byte in ACCA. This is the routine used for tag entry in the debugger; see "Entering a tag" for details of entering a number. If ESC is typed during the entry of the hex characters, HEXIN immediately returns to the calling routine with the V condition code set, and undefined values in IX and ACCB and ACCA. If entry is terminated normally, with a CR, the V condition code is clear on return. This permits the calling routine to test whether the user aborted the entry.

DEBUGR 31 $1F Debugger

Transfers control to the debugger with memory mode off, output to screen, and upper-case flag set.

Miscellaneous functions

LINEIN 32 $20 Line input

Inputs a character string from the keyboard, and stores it starting at the address in LINBUF. For complete explanation, see "GMXBUG line input utility".

MKWRIT 33 $21 Write MIKBUG-format tape file

Writes the contents of the bytes from the address in BEGIN1 through the address in END1 inclusive to cassette tape in MIKBUG format. For each record written a "#" is displayed. No device control characters are generated, so the tape must be started and stopped by hand. See "MIKBUG file format" for details of the MIKBUG format.

MKREAD 34 $22 Read MIKBUG-format tape file

Reads a MIKBUG format file from tape into memory. For each record read in a "#" is displayed. If the checksum of a record is in error, the block address of that record is displayed with a "?", and the read continues. If the second byte of TEMP ($A010) is 00, the data in each record is stored starting at the address in that record's heading. If the second byte of TEMP is not 00, then all the data in the file is stored in one continuous block, starting at the address that is in BEGIN2 when the read starts. When the S9 at the end of the file is read, the address where the last byte of data was stored, plus one, is left in BEGIN2 to point to the end of the file. See "MIKBUG file format" for details of the MIKBUG format.

PCR 35 $23 Print carriage return

Sends a carriage return to OUTCHR. For details, see "GMXBUG video driver".

PLF 36 $24 Print line feed

Sends a line feed to OUTCHR. For details, see "GMXBUG video driver".

INECHO 37 $25 Input with echo

Calls INCHAR to input a character from the keyboard, calls OUTCHR to echo the character, and returns the input character in ACCA.

INKEY 38 $26 Input from keyboard

Scans keyboard interface for "DATA READY" signal. If data is present, returns the Z condition code clear and the input byte in ACCA. If no data is present, returns with the Z condition code set.

FDB INKEY.
BEQ → no inp
BNE → input.

PRINT 39 $27 Print character on auxiliary printer

Outputs the byte in ACCA either to a parallel printer connected to port #4 B side, or a serial printer connected to port #3. If PRFLAG ($A03E) is 0, the parallel port is used; if PRFLAG is not 0, then the serial port is used. If the byte is a CR, it is followed by null characters to allow the carriage to return on unbuffered printers. The number of nulls sent is kept in NULCNT ($A037), and may be set using the "I" command. It is initialized to 6.

TBSRCH 40 $28 Table search

Indexes through a table of the form

```
FCB   ID1
FDB   DATA1
FCB   ID2
FDB   DATA2
.
.
.
FCB   IDN
FDB   DATAN
FCB   00
```

The start of the table must be in IX, and the ID byte must be in ACCA. If the ID byte is matched, the data field is returned in IX, and the Z condition code is returned set. If there is no match, then Z is returned clear. The end of the table must be marked by an ID byte of 0. 0 is an illegal ID byte, and always returns "no match".

ESCSND 41 $29 Send escape code

Sends an ESC to the video driver followed by the character in ACCA.

INCAPS 42 $2A Input upper case only

Inputs a character with the force upper case flag on.

GMXBUG line input utility

This utility is called LINEIN and is system call 32. It inputs characters from the keyboard, echoes them on the screen, and stores them in the line input buffer. It can handle strings up to 80 characters long, and recognizes three control keys.

The line input buffer is the area of memory pointed to by LINBUF ($A01F). LINBUF is initialized by the reset routine to point to the highest 256 bytes below $6FFF, but the user may change LINBUF to point to any address which is an even multiple of 256. As characters are typed on the keyboard, they are stored in successive locations starting at the address in LINBUF. The pointer to the next empty byte is called ENDLIN ($A01D). Only printing characters are stored, except for CR, which is stored at the end of the string. On return from LINEIN, ENDLIN points to the byte after the the CR. If LINBUF is set correctly, then on return the second byte of ENDLIN ($A01E) will have the count of characters entered including the CR.

LINEIN recognizes three control keys: BACKSPACE (ctl-H), ESC, and CR.

BACKSPACE moves the cursor one space to the left on the screen, erases the last character typed, and moves ENDLIN back one byte in the buffer. If ENDLIN already points to the first byte in the buffer, then BACKSPACE has no effect.

ESC terminates input and returns to the calling program with ESC in ACCA. ESC is for aborting entry.

CR terminates input, stores a CR in the buffer to terminate the string, and returns to the calling program with CR in ACCA. CR is for completing entry.

GMXBUG video driver

One of the advantages of the GMXBUG monitor is that instead of using a terminal as the console output device, it uses the Gimix video board as the console output device. This eliminates the need for a terminal, and also eliminates the time required by limited speed terminals to handle the output. The video board operates at the speed of the system, displaying characters as fast as the CPU can write data to memory.

The video board has a 2K x 8 static memory on board. This memory is acessible from the bus like any other memory. This is where the characters displayed on the screen are stored. The video board also has circuits which do a continuous sequential read of the 2K memory. As each byte is read, it is fed to the character generator and control circuits. These circuits convert the character code to the dot pattern stored in one of the three character generators. This pattern is modified according to the mode settings for half intensity and inverse video, then encoded into the video signal by the video generator, thus displaying the character on the screen.

The timing of the sequential read is synchronized with the timing of the video signal, so that the character in the first byte of the 2K memory appears at the upper left corner of the screen, the character in the second byte just to the right of the first, the character in the eightieth byte in the upper right corner, the character in the eighty-first byte directly under the first, and so on. The last 128 bytes of this memory are not used.

The GMXBUG console output routine, OUTCHR, which is system call 17 ($11), allows the user to treat the video board as a "glass teletype". OUTCHR performs all the necessary control functions to convert the character stream output into an organized display.

To use OUTCHR, load the character to be displayed into ACCA and use system call 17. The character will be displayed. ACCA, ACCB, and IX will not be altered.

Before getting into the details of the video driver, please read the "Gimix Super Video Board User's Guide", which describes all the hardware functions of the video board.

Pointers and registers

OUTCHR maintains pointers into the video board memory, and also certain hardware registers on the video board itself. The names and functions of these pointers and registers are given in the following table:

CURSOR A018 pointer to the byte where the next printing character will be stored.

WRAP	A01A	counts the number of spaces available on the current line.
CHRSLT	A01B	mask for switching between display of slot 0 and slot 1 characters. Each character stored in the screen memory is ORed with CHRSLT.
ESCSTR	A01C	counter for number of bytes expected in an escape sequence.
ESCVEC	A03B	vector to currently active escape routine.
SCROLL	F901	number of line which appears at the BOTTOM of the screen.
XCRSR	F902	cursor column position.
YCRSR	F903	cursor row position.

When a printing character is received by OUTCHR, it is stored in the byte pointed to by CURSOR. CURSOR is incremented, and WRAP is decremented. If CURSOR reaches $F780 (1 past the end of the last line) it is set to $F000. If WRAP reaches -1 it is set to 79. XCRSR is incremented; if it reaches 80 then it is set to 0 and YCRSR is incremented. If YCRSR reaches 24 it is set to 0.

Control characters

OUTCHR recognizes five control characters: BELL ($07), carriage return ($0D), line feed ($0A), form feed ($0C), and ESC ($1B). All other control characters are ignored. The effects of these five characters are as follows:

BELL : the 1-bit latched output is toggled rapidly for a short time. If the latch is connected to a speaker, this produces an audible beep tone. The duration and pitch of the beep are determined by DURATN ($A03F) and PERIOD ($A041). DURATN is the duration of the beep in "ticks", a tick being 16 machine cycles. PERIOD is the interval between toggles in ticks. PERIOD and DURATN are initialized for a 1042 Hz tone for .25 seconds, but the user may change them to produce different beeps, or even play simple tunes.

Carriage return: 80 is subtracted from the value in WRAP, and the result is negated and subtracted from CURSOR. This sets CURSOR to the start of the line, unless the last printing character was displayed in the last column of the previous line, causing CURSOR to wrap around. In that case, CURSOR is set to the start of the previous line. XCRSR is always set to 0, WRAP is always set to 80. YCRSR is not changed

unless WRAP is 0, indicating wraparound, in which case YCRSR is decremented.

Line feed : CURSOR is increased by 80; that is, CURSOR is changed to point to the position immediately below the one it originally pointed to. YCRSR is incremented, unless it was 23, in which case it is set to 0. XCRSR and WRAP are not affected.

Form feed : CURSOR is set to $F000, XCRSR and YCRSR are set to 0, WRAP is set to 80, SCROLL is set to 23, and $20 is stored in every byte of the screen memory. This moves CURSOR to the upper left corner, and erases the screen by filling it with spaces.

ESC : enables the escape sequence processor. See "Escape sequences" for details.

DELETE

The DELETE character ($7F) is included in the ASCII character set for masking erroneous characters on paper tape. It is not a control character, but not a printing character either. OUTCHR ignores DELETE characters.

Scrolling

Each time OUTCHR increments YCRSR, it compares YCRSR to SCROLL. If YCRSR is equal to SCROLL before incrementing, then the bottom of the screen has been reached and scrolling must be performed. SCROLL is incremented (set to 0 if 23), and the new bottom line is filled with spaces.

To avoid annoying "video splitting" of the image, OUTCHR waits for a vertical sync interval before incrementing SCROLL. Since vertical sync occurs 60 times a second, this limits scrolling to 60 lines a second. A stream of line feed characters would be processed at 60 characters per second.

Output stop/start

When more than 23 lines of output are sent to the video board at once, such as a listing of a BASIC program, the first part of the output may scroll up off the top of the screen before the user gets a look at it. Therefore a facility for momentarily halting output has been provided. To halt a stream of output, type ctl-S. This halts the output, and puts the system into a loop of testing the keyboard interface until another ctl-S is typed. Then the system leaves the loop and continues with what it

was doing. After the first ctl-S has been typed only another ctl-S will have any effect; all other characters will be ignored.

The ctl-S function may be enabled or disabled using the "I" command.

Escape sequences

Because the Gimix 80 x 24 video board has so many special features, the video driver has been designed to permit control of these features by command strings in the ASCII output stream. Such a command string is indicated by an ESC control character ($1B). When the video driver gets an ESC, the ESCSTR flag is set to 1 to indicate that the escape sequence processor is active, and the address of the escape code handler is put in ESCVEC. The next time the video driver is called, the byte in ACCA is used as an index to get the address of the escape operation which is then executed, and ESCSTR is then cleared, shutting down the escape processor. If the code byte is greater than 21 ($15) or is 0, then ESCSTR is cleared with no action taken.

Some escape sequences contain additional bytes of data. In such cases the escape processor is left active for as many bytes as the sequence contains. Two escape sequences (PRGMOD and TRMODE) have a count as the third byte of the sequence. This count is placed in ESCSTR. On each subsequent call to OUTCHR, the data byte is directly passed to the escape routine, and ESCSTR is decremented, until ESCSTR reaches 0, shutting down the escape processor.

Escape code descriptions

Name	code #	Format	Function
POSX	1	ESC-1-#	Sets XCRSR and CURSOR so that CURSOR points to column # of the current row.
POSY	2	ESC-2-#	Sets YCRSR and CURSOR so that CURSOR points to row # at the current column.
MEMOFF	3	ESC-3	Disables the video board memory.
MEMON	4	ESC-4	Enables the video board memory.
SLCVID	5	ESC-5	Selects the screen memory.
SLCFNT	6	ESC-6	Selects the font memory.
SETM0	7	ESC-7	Set MODE 0.
SETM1	8	ESC-8	Set MODE 1.

VDBLNK	9		ESC-9	Blanks video image.
VDNORM	10	A	ESC-10	Turns on video image.
CBOFF	11	B	ESC-11	Disables hardware cursor.
CBON	12	C	ESC-12	Enables hardware cursor.
CBSTDY	13	D	ESC-13	Selects steady cursor.
CBFLSH	14	E	ESC-14	Selects blinking cursor.
OUTSCR	15	F	ESC-15	Routes output to video screen.
OUTPRT	16	10	ESC-16	Routes output to aux printer.
OUTBTH	17	11	ESC-17 Q	Routes output to both screen and printer.
SLOTO	18	12	ESC-18 R	Sets OR-mask for characters to 0, causing SLOT 0 characters to be displayed.
SLOT1	19	13	ESC-19 S	Sets OR-mask to $80, causing slot 1 characters to be displayed.
PRGMOD	20	14	ESC-20-#-	Stores the next # bytes in the mode control memory.
TRMODE	21	15	ESC-21-#-	Causes the next # characters to be displayed in "transparent mode": all characters are stored in the screen memory, including control chacters.

MIKBUG file format

GMXBUG includes routines for reading and writing tape files in the "MIKBUG" (TM) format. This format is used by most 6800-based systems, and was invented by Motorola. In this format the data is written to the tape in blocks which are called records, and each data byte is converted to two ASCII characters representing the byte in hexadecimal.

There are three types of records in the MIKBUG format: header records, data records, and end-of-file records. Header records start with "S0" and contain a name or other identifier for the file. GMXBUG does not generate or recognize header records; if a file containing a header record is read by GMXBUG the header is ignored and has no effect.

Data records begin with "S1". The next two characters are the byte count in hexadecimal. The next four characters are the address in hexadecimal. Each pair of characters after that is a data byte, except the last pair, which is a checksum. The byte count is the number of data bytes in the record plus two for the address plus one for the checksum; if there were five bytes of data the byte count would be eight. The address is the starting address of the area where the data will be stored as the record is read in. The checksum is the 1s complement of the modulo 256 sum of the byte count, the address, and the data bytes. This value is calculated by GMXBUG as the record is read, and checked against the value in the record; if they do not match, GMXBUG reports an error. A record looks like this:

S10701230102O3CE where 07 is the byte count, 0123 is the address, 010203 is the data, and CE is the checksum.

An end-of-file record consists of "S9". When this is read the tape load stops. GMXBUG places "S9" at the end of all files.

The characters after a checksum and before the following "S1" are ignored by GMXBUG when reading a file. When writing a file, GMXBUG puts a CR, LF, and two null characters after each checksum. This is so that if a GMXBUG-written file is read using the tape reader of an ASR type terminal, the print image or screen image of the tape will come out right.

Patches

These patches are provided for users who wish to adapt software from other companies to run under GMXBUG.

Implementing the patches

For programs supplied on K.C. standard cassette, the procedure for implementing patches is: load the program from cassette into memory, make the changes using the GMXBUG M command, then save the patched version on cassette.

If the program is supplied on FLEX compatible diskette, the procedure for implementing the patches is: type "GET,<filename.ext>" to move the program into memory without running it, then type "MON" to go to GMXBUG. Make the changes using the GMXBUG M command, then re-enter FLEX at $7103 by typing "G (CR)". Then type "SAVE,<modfile.ext>,<starting address>, <ending address>,<transfer address>" to save the patched version on disk.

If the program is supplied on S.S.B. compatible diskette, then the procedure is: type "GET,<filename.ext>" to load the program to be patched. Exit from the DOS to GMXBUG to make the patches using the GMXBUG M command. Then return to the DOS with a warm start, and type "SAVE,<newname.ext>,<starting address>, <ending address>,<transfer address>" to save the patched version on disk.

Duplex

Many interactive programs are set up to run in half duplex, while GMXBUG is set up to run in full duplex. This conflict can be resolved in two ways: first, the I command can be used to set the console duplex to half before starting the program, or second, the program can be patched to force full duplex operation. The patches marked "**" are optional patches which force full duplex operation.

SWTPC 8K BASIC 2.0

at	replace	with	purpose
010F	7E E1D1	7E 03CC	use port #1 as standard ACIA
	7E E1AC	7E 03BD	
	7E 171F	7E 0347	
0121	7E 03CC	3F 27	optional: use GMXBUG printer
		39	driver to drive port #4
	7E 03BD	7E 171F	
	7E 0347	7E 171F	
012A	7E 03E2	7E E1D1	tie port #4 to console

	7E 03D7	7E E1AC	
	7E 0352	7E 171F	
012D	7E E1AC	3F 25 39	force keyboard echo **
0150	01	04	move control port to port #4
0154	0F 5F	08 00	modify for GMXBUG backspace
03A3	B6 8004 2B 12	3F 26 01 27 12	make ctl-C test compatible with GMXBUG keyboard interface
03AD	BD E1AC	01 01 01	part of above
0C40	8C 8000	BC A01F	use LINBUF as memory limit

Note: since the control port is the console, and therefore not connected to the cassette interface, programs must be saved and loaded through port #0, which is equivalent to the GMXBUG cassette interface.

SWTPC 8K BASIC 2.2

at	replace	with	purpose
010F	7E E1D1 7E E1AC 7E 1725	7D 03CC 7E 03BD 7E 0347	use port #1 as standard ACIA
0121	7E 03CC 7E 03D7 7E 0352	3F 27 39 7E 1725 7E 1725	optional: use GMXBUG printer driver to drive port #3
012A	7E 03E2 7E 03D7 7E 0352	7E E1D1 7E E1AC 7E 1725	tie port #4 to console
012D	7E E1AC	3F 25 39	force keyboard echo **
0150	01	04	move control port to port #4
0154	0F 5F	08 00	modify for GMXBUG backspace
03A3	7E 1ECB 2B 12	3F 26 01 27 12	make ctl-C test compatible with GMXBUG keyboard interface
03AD	BD E1AC	01 01 01	part of above
0B91	BD 1EB0	7F 0091	eliminate call to MP-C/MP-S

			interface compatibility routine
0C40	8C 8000	BC A01F	use LINBUF as memory limit

Note: since the control port is the console, and therefore not connected to the cassette interface, programs must be saved and loaded through port #0, which is equivalent to the GMXBUG cassette interface.

SWTPC 8K BASIC 2.3

at	replace	with	purpose
0106	7E 1E42	7E 0412	use port #0 as standard ACIA
	7E 1E36	7E 0403	
	7E 1E2D	7E 037B	
010F	7E 015C	7E 0412	use port #1 as standard ACIA
	7E 0167	7E 0403	
	7E 036C	7E 037B	
0121	7E 03CC	3F 27	optional: use GMXBUG printer
		39	driver to drive port #3
	7E 0412	01 01 39	
	7E 037B	01 01 39	
012A	7E 044A	7E 015C	tie port #4 to console
	7E 043F	7E 0167	
	7E 0387	7E 036C	
0150	01	04	move control port to port #4
0154	0F	08	modify for GMXBUG backspace
	5F	00	
015C	7D 001F	3F 11	convert MP-C/MP-S adapter
	26 03	7E 041C	routine to GMXBUG interface
	7E 0412	01 01 01	
	7E E1D1	01 01 01	
	7D 001F	3F 10	
	26 03	20 F1 01	
	7E 0403	01 01 01	
	7E E1AC	01 01 01	
036C	7F 001F	7F 001F	make sure MP-S/MP-C
	B6 8004	39 01 01	flag always indicates
			MP-S type interface
03D8	96 1F	01 01	make break test compatible
	26 13	01 01	with GMXBUG keyboard
	B6 8004	3F 26 01	interface
	47	01	
	24 0A	27 0A	
	B6 8005	01 01 01	

041C	E6 00	3F 26	make ESC test compatible
	47	01	with GMXBUG keyboard
	24 17	27 17	interface
	A6 01	01 01	
042A	8D D7	3F 10	part of above
0CAB	8C 8000	BC A01F	use LINBUF as memory limit

Note: since the control port is the console, and therefore not connected to the cassette interface, programs must be saved and loaded through port #0, which is equivalent to the GMXBUG cassette interface.

Note: the original code for the break test assumes that output is being sent to one of the four serial ports, and accepts an ESC from that port only. The patched code assumes that output is being sent to one of the four serial ports or to port #4 (the console), and will only accept ESC from the console, even if the control port is changed.

SWTPC DISK BASIC 3.0

at	replace	with	purpose
0106	7E 2224	7E 0476	convert port #0 to standard ACIA
	7E 2218	7E 0467	
	7E 220F	7E 03B2	
010F	7E 7112	7E 0476	convert port #1 to standard ACIA
	7E 710F	7E 0467	
	7E 198C	7E 03B2	
0121	7E 0476	3F 27	optional: use GMXBUG printer
		39	driver to drive port #3
	7E 0467	01 01 39	
	7E 03B2	01 01 39	
012A	7E 048C	7E 7112	tie port #4 to DOS concole
	7E 0481	7E 710F	
	7E 03BD	7E 198C	
0150	01	04	move control port to port #4
0440	B6 8004	3F 26 01	make ctl-C test compatible with
	46	01	GMXBUG keyboard interface
	24 1E	27 1E	
	B6 8005	01 01 01	
0527	81 04	81 10	fix test for DOS console port

starting address: $0100
ending address : $23D0
transfer address: $0100

Note: under SWTPC disk BASIC 3.0, port #0 is defined as an MP-C "bit-banger" type interface, and special I/O drivers are provided for this port. These drivers will only operate under SWTBUG! Any attempt to use an MP-C interface with these drivers under GMXBUG will bomb!

FLEX DOS 1.0 (mini-FLEX)

Two changes must be made to MINIFLEX to make it fully compatible with GMXBUG 3.0. First, the keyboard test for break characters must be modified for the GMXBUG keyboard interface, and second, the setup of the stack area in the scratchpad must be modified to set the stack at the top of user memory. Also, the user may want the PRFLAG and UPCASE flags modified when FLEX starts up, and if the MON command is used, GMXBUG should be entered without performing the RESET routine. These modifications can be made to FLEX by creating the following program as a utility command and setting up a "STARTUP" file to invoke it when FLEX is booted up.

```
                        * startup modifier for FLEX
A01F                    LINBUF  EQU     $A01F
0004                    ADDAX   EQU     4
0026                    INKEY   EQU     38
7600                            ORG     $7600
                        * Fix stack setup
7600 FE A0 1F           START   LDX     LINBUF
7603 86 FF                      LDA A   $FF
7605 3F                         SWI
7606 04                         FCB     ADDAX
7607 FF 71 46                   STX     $7146
760A FF 71 5E                   STX     $715E
                        * Fix monitor address
760D CE E0 E3                   LDX     $E0E3
7610 FF 70 F5                   STX     $70F5
                        * Set printer type to serial  (optional)
7613 B7 A0 3E                   STA A   PRFLAG
7616 7E 71 03                   JMP     $7103
                        * fix for keyboard test
72E2                            ORG     $72E2
72E2                            SWI
72E3                            FCB     INKEY
72E4                            NOP
72E5                            NOP
72E6 26 0D                      BNE     *+15
72E8                            NOP
72E9                            NOP
72EA                            NOP
                                END     START
NO ERROR(S) DETECTED
 SYMBOL TABLE:
ADDAX  0004    INKEY  0026    LINBUF A01F    START  7600
```

miniFLEX PRINT.SYS

The PRINT.SYS driver supplied with miniFLEX is not compatible with GMXBUG, since it resides in the scratchpad. Also, it is redundant to have a separate printer driver besides the one provided by GMXBUG. The following program replaces the TSC PRINT.SYS and avoids these problems.

```
                    * system printer driver for miniFLEX
                    *
0027                PRINT   EQU     39
                    *
 0010                       ORG     $10
 0010 70 7F                 FDB     RETURN
 0012 70 7D                 FDB     DOPRNT
                    *
                            ORG     $707D
 707D 3F            DOPRNT  SWI
 707E 27                    FCB     PRINT
 707F 39            RETURN  RTS
                    *
 710D                       ORG     $710D
 710D 70 7D                 FDB     DOPRNT
                    *
                            END

NO ERROR(S) DETECTED

 SYMBOL TABLE:

DOPRNT 707D   PRINT  0027   RETURN 707F
```

TSC Text Editor (FLEX disk V1.0)

at	replace	with	purpose
020C	7E E1AC	3F 12 39	set tape input from tape input routine
020F	7E E1D1	3F 13 39	send tape output to tape output routine
1661	8E 01FF	01 01 01	leave stack where it is
169D	8C 7000	BC A01F	use LINBUF as memory limit

```
starting address: $00B1
ending address  : $19DA
transfer address: $0200
```

TSC Text Processor (FLEX disk V1.0)

at	replace	with	purpose
020C	8E 01FF	01 01 01	leave stack where it is
1595	B6 70A1	3F 26	make break test compatible with
	27 06	27 04	GMXBUG keyboard interface
	B6 8004	81 03	
	44	27 01	
	25 01	39	
	39	CE 1714	
	B6 8005	7E 0D0E	

starting address: $0200
ending address : $1779
transfer address: $0200

TSC Assembler (FLEX disk V1.0)

at	replace	with	purpose
0323	7E E1D1	3F 13	send tape output to
		39	tape output routine
1602	8E A07F	01 01 01	stack should not be set up in scratch pad
1636	8C 7000	BC A01F	use LINBUF as memory limit

starting address: $0300
ending address : $19AB
transfer address: $0300

TSC Text Editor

at	replace	with	purpose
0206	7E E1AC	BD E1AC	force keyboard echo **
020C	7E E1AC	3F 12	send tape output to tape
		39	output routine
020F	7E E1D1	3F 13	set tape input from tape
		39	input routine
0355	8E 01FF	01 01 01	leave stack alone
038A	8E 01FF	01 01 01	leave stack alone

TSC Text Processor

at	replace	with	purpose
0206	7E E1AC	3F 25 39	force keyboard echo **
1471	B6 8004	3F 26 01	make break test compatible with GMXBUG keyboard interface
	2A 01	26 01	
	39	39	
	B6 8004	01 01 01	
	2A FB	01 01	
	01 01 01	01 01 01	

TSC Assembler V1.4

at	replace	with	purpose
031B	7E E0D0	7E E0E3	no screen erase on return from the assembler.
0323	7E E1D1	3F 13	sends object code to tape inter-
		39	face.

The assembler listing is outputted through OUTEEE, and appears on the console device (the GMXBUG video screen). If a hard copy listing is desired, the GMXBUG I command can be used to send console output to the printer.

TSC MICRO-BASIC PLUS V2.1

at	replace	with	purpose
0109	7E E1AC	3F 25 39	force keyboard echo **
01B6	8E A07F	01 01 01	stack should not be set in scratch pad
0452	36	3F 26	make break routine
	B6 8004	26 01	compatible with GMXBUG
	2A 02	39 01	keyboard interface.
	32	01	
	39	01	
	B6 8004	01 01 01	
	2A FB	01 01	
0945	CE A023	FE A01F	use LINBUF value as stack limit

TSC SPACE VOYAGE

at	replace	with	purpose
00FB	7E A04A	7E 1000	move random routine out of scratch pad
0100	8E A042	01 01 01	leave stack where it is

RANDOM is the TSC random number generator, and is set up by TSC in the 128-byte scratchpad. Since using the scratch pad is undesirable under GMXBUG, the following code must be appended to SPACE VOYAGE to provide this routine.

Note: $1027-$102A must not all be zero at start.

```
                    * additional code for TSC SPACE VOYAGE
                    *
 1000                         ORG    $1000
                    *
                    * this code is the TSC random number generator
                    * with the origin changed to $1000
 1000 F7 10 26  RANDOM  STA B  BSAVE
 1003 C6 08             LDA B  #8
 1005 B6 10 2A  RPT     LDA A  RNDM+3
 1008 48                ASL A
 1009 48                ASL A
 100A 48                ASL A
 100B B8 10 2A          EOR A  RNDM+3
 100E 48                ASL A
 100F 48                ASL A
 1010 79 10 27          ROL    RNDM
 1013 79 10 28          ROL    RNDM+1
 1016 79 10 29          ROL    RNDM+2
 1019 79 10 2A          ROL    RNDM+3
 101C 5A                DEC B
 101D 26 E6             BNE    RPT
 101F F6 1026           LDA B  BSAVE
 1022 B6 1027           LDA A  RNDM
 1025 39                RTS
 1026           BSAVE   RMB    1
 1027           RNDM    RMB    4

                        END

NO ERROR(S) DETECTED

 SYMBOL TABLE:
BSAVE 1026   RANDOM 1000   RNDM   102A   RPT    1005
```

TSC Relocator

at	replace	with	purpose
02A0	86 3C	20 08	tape can not be controlled by software

TSC Debug Package

Patching the Debugger

The debugger directly accesses the control interface, assuming it to be an ACIA at port #1. This must be patched to access the GMXBUG control interface.

Also, the nesting level counter in the debugger is fooled by the RTS in the SWI processor; it decrements on that RTS, thus being thrown off. Eventually, an "RTS in level 0" will occur, aborting the simulation. If the SWI call is not in a called subroutine, this error occurs immediately.

Fixing the keyboard interface problem requires the following two patches:

at $42B6,	replace	with
	FE 410F	7F 408A
	7F 408A	3F 10
	A6 00	84 7F
	46	B1 4114
	24 FB	26 EC
	A6 01	81 03
	84 7F	27 04
	B1 4114	81 0D
	27 E3	26 F0
	81 03	7E 4195

Note: this will create an 8-byte empty space, which will be used later.

Also, at $449E,	replace	with
	FE 401F	3F 26 01
	A6 00	01 01
	46	01
	24 46	27 46
	A6 01	01 01

Fixing the nesting counter problem is a little more difficult. The solution is to increment the nesting counter whenever an SWI is executed. The RTS in the SWI processor is thus compensated for. To make this fix, the following the following patches must be made:

at $4814,	replace	with
	FE 4115	BD 42CD

and at $42CD, which is the 8-byte space made by the first patch, insert

```
7C 4099
FE 4115
39
01
```

These last two patches have a minor side effect. The trace function is controlled by the nesting level counter and the "T" register. Code whose nesting level is less than the value in "T" will be traced. If the nesting level when the SWI call is simulated is one less than the value in "T" then the SWI processor will not be traced. Tracing will resume with the first instruction of the called routine.

SSB Super Editor

at	replace	with	purpose
0206	7E E1AC	BD E1AC	force keyboard echo **
020C	7E E1AC	3F 12 39	set tape input from tape input routine
020F	7E E1D1	3F 13 39	send tape output to tape output routine
038A	8E 01FF	01 01 01	leave stack alone
1683	8E 01FF	01 01 01	leave stack alone

```
starting address: $0000
ending address  : $1957
transfer address: $1683
```

SSB Super Assembler

at	replace	with	purpose
0300	8E A07F	01 01 01	leave stack alone

```
0323    7E E1D1     3F 13 39     send tape output to
                                 tape output routine

1632    BD E1D1     3F 27 39     set up hard copy
        7E E1D1     39 01 01

starting address: $0300
ending address  : $17A2
transfer address: $1615
```

GRT G2 BASIC 1.0

Note: G2 BASIC is recorded in a special non-standard binary format. A special loader program is required to read it into memory. This loader is recorded on the tape in MIKBUG format ahead of the BASIC. The loader must be read in and patched before the BASIC can be read in. The patches for the loader are as follows:

```
 at     replace     with         purpose

1F0A    27 0D       20 0D        eliminate serial/MP-C test

1F21    B7 8004     B7 8000      move serial interface to port #0

1F2A    7E 0000     3F 1F 01     return to debugger after loading BASIC

1F6F    CE 8004     CE 8000      move serial interface to port #0

starting address: $1F00
ending address  : $1FA8
transfer address: $1F00
```

The patches for G2 BASIC itself are as follows:

```
 at     replace     with         purpose

0478    BD 1482     3F 10 01     keyboard input fix

067B    BD 14D8     3F 26        keyboard test fix
        24 FA       26 04
                    39

13EA    BD 1482     BD 1478      tape input fix

1478    B6 8004     B6 8000      move tape interface to port #0
        46          46
        24 FA       24 FA
        B6 8005     B6 8001

14AD    8D F9       84 7F        console output fix
        C6 0A       3F 11
```

	7A 8004	32 39	
1332	BD 14AC	3F 13 01	tape output fix
0000	7E 14E3	01 01 01	eliminate erroneous initialization
1CFB	B6 8005	7E 1D1B	eliminate erroneous initialization

```
starting address: $0000
ending address  : $1ED2
transfer address: $0000
```

Note: G2 BASIC has a special command and a special statement for using an auxiliary printer (LLIST and LPRINT). LLIST and LPRINT use G2 BASIC's internal printer driver, and instructions for patching this driver are included in the G2 BASIC manual. However, since GMXBUG has the facility for dynamically switching console output between the video screen and an auxiliary printer under program control, the use of these statements is unnecessary with GMXBUG. Instead, the user can use the following BASIC statements and commands to get hard copy when he wants it. A further advantage of this is that a program's output may be changed from screen to printer or vice versa without the labor of replacing every LPRINT or PRINT statement.

To switch the output of a program to the printer, use

PRINT CHR$(27);CHR$(16) or POKE 41014,255

and to switch from the printer to the screen, use

PRINT CHR$(27);CHR$(15) or POKE 41014,0

To list the program on the printer, use

PRINT CHR$(27);CHR$(16); : LIST [line #] : PRINT CHR$(27);CHR$(15)

or

POKE 41014,255 : LIST [line #] : POKE 41014,0

PERCOM miniDOS 1.2

at	replace	with	purpose
C2EA	8E A042	01 01 01	leave stack alone

Note: PERCOM miniDOS is supplied on a 2708 EPROM. To implement this patch, install the PERCOM controller card with the EPROM in your system, then copy the EPROM contents to RAM. Make the patch

using GMXBUG. Then use a 2708 programmer such as the Gimix P.P.D. To program a new 2708 with the patched version version of miniDOS. You can keep the old EPROM as a backup, or erase it and use for another purpose.

PERCOM TOUCHUP for TSC EDITOR

at	replace	with	purpose
16DB	CE 1D7B	FE A014	use LINBUF as memory limit
	6F 00	20 0A	
1CC3	DF 22	3F 27	convert to GMXBUG print driver
	FE 15F0	20 26	
1CED	37	3F 11	convert for GMXBUG type console
	F6 15F3	3F 27	
	2A 0E	26 26	
	7F 15F3	39	

PERCOM Assembler

at	replace	with	purpose
0109	7E E1D1	BD E1D1	force keyboard echo **
0120	8E A07F	01 01 01	leave stack alone
0307	DF DD	3F 27	convert to GMXBUG print driver
	FE 010F	20 24 01	
0330	B6 8004	3F 26	convert break test for GMXBUG console
	B1 8006	27 37	
		20 1A	

TSC BASIC

at	replace	with	purpose
0109	7E 011E	3F 25 39	convert to GMXBUG console I/O
010F	7E 0132	3F 11 39	convert to GMXBUG console I/O
0112	7E 0109	3F 12 39	convert to GMXBUG tape I/O
0115	7E 010C	3F 13	convert to GMXBUG tape I/O
01C7	DE 42 A6 00 84 01 27 08 01 01	3F 26 01 01 01 01 27 08 01 01	convert ctl-C test for GMXBUG console
01F8	DE 42 A6 00 84 01	3F 26 01 01 01 01	part of above
0252	DE 42 A6 00 84 01	3F 26 01 01 01 01	part of above